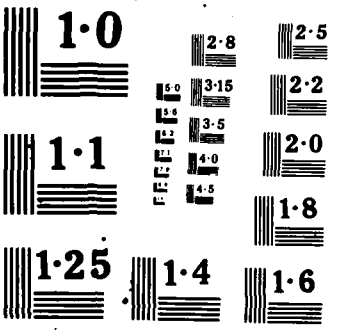
AD-A168 085 ORWELLIAN PROGRAMMING IN SAFETY-CRITICAL SYSTEMS(U) 1/1
ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND)
I F CURRIE 1984 RSRE-MEMO-3924 DRIC-BR-99100
UNCLASSIFIED F/G 9/2 NL



RSRE MEMORANDUM No. 3924

ROYAL SIGNALS & RADAR ESTABLISHMENT

ORWELLIAN PROGRAMMING IN SAFETY-CRITICAL SYSTEMS

Author: I F Currie

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

Royal Signals and Radar Establishment
Memorandum No. 3924

ORWELLIAN PROGRAMMING IN SAFETY-CRITICAL SYSTEMS

Ian F Currie

Computers are being used increasingly in situations where lives will be at hazard if they fail. Failures resulting from software errors are both avoidable and unforgivable. This paper is concerned with the final refinement in the production of a critical program from an elementary language via a compiler to bits in some rom in the critical system. It argues that existing languages are inherently unsuitable for this purpose since they allow freedoms of expression and thought which, although convenient in some applications, lead to misconceptions and obscurities. Orwellian programming should limit one's freedom of expression so that these heretical tendencies are unthinkable by using a NewSpeak which permits only Good Thoughts.

This paper was originally given at a conference sponsored by the International Federation of Information Processing and the International Federation of Operational Research Societies on "System Implementation Languages, Experience and Assessment" held at Canterbury University in August 1984.

What are safety-critical systems?

A safety-critical system is one where people are likely to die if the system fails for whatever reason. Examples are the controlling operations in aircraft or nuclear reactors in which computer control is playing an increasing role. Computer control is necessary either because human reaction times are too slow or because the human attention span is too limited for vital (but boring) monitoring of safety instruments. Clearly, the computers and their programs are still only a small part of the total system at risk - the total system will include sensors, analogue to digital converters, actuators etc. However, there are far more opportunities for making obscure errors in programs than in the rest of the system and the way that faults occur in program make nonsense of the kind of calculations used to estimate error rates in hardware. Also, the redundancy necessary to guard against hardware failure is useless against software (or firmware) errors; programming errors are usually "common mode" errors.

Most safety critical programs operating at the moment are simple control programs; the computer continually reads a set of sensors and continually produces a set of control signals depending on some predefined algorithm. Such a control program is likely be written as a loop repeating the same calculations indefinitely. In general this loop must be repeated within some specified time interval.

As an example, consider a "fly-by-wire" aircraft. The sensory input to the controlling program includes instructions from the pilot's controls, air-speed indicators etc; the outputs are instructions to the contol surfaces (flaps etc). The algorithm connecting the inputs to the outputs depends on the aerodynamic properties of the airframe as does the sampling interval given by the time round the loop. For a particular aircraft there will be an upper limit specified for this sampling time depending on the intrinsic stability of the aircraft and will typically be between 1 and 100 msecs.

The complexity of programs in safety critical situations is bound to increase. Furthermore, our perception of what is safety critical could change as a result of environmental changes, military action or political manoeuvres. For example, the programming of in-flight navigational computers has not in the past been considered highly safety critical. Perhaps the downing of the South Korean airliner will alter this.

How is safety ensured?

In any safety critical system, the complete system and each subsystem down to the component level is very heavily tested both in simulation test-rigs and live operation. The process of certifying the system is one of gaining sufficient confidence in its operation derived both from the results of these tests plus a great deal of more abstract investigation of the design of the system. Since testing a system can only give a definite result if the system is, in fact, faulty, the abstract investigation is just as important as the test-rigs. I hesitate to use the word "proving" in this context since, for example, it is difficult to imagine a formal proof for an analogue-to-digital converter.

Evaluating the total system can require expertise from many different fields. For example, to evaluate properly a system for the computer control of the the flaps of an aircraft, the certifying authority has to use the skills of at least aerodynamicists, control engineers, electronic engineers, computer programmers and test pilots. The picture is one of succesive layers of expertise being applied in each field to generate sufficient confidence in the whole. Thus, the aerodynamicist says "If you move the flaps this way, you will get this effect and you won't tear them off the plane"; the control engineer says "Given a control signal here, the flaps will move thus"; the electronic enginers say "The computer input is given by the sensors and its output gives the control signals"; the computer programmer says " The inputs and the outputs of the computer are related thus" and the test pilot says "When I do this, the flaps appear to do that". Each specialist has to ensure that specifications are met within his own field, and that the interfaces into other specialist areas are consistent. An overview of all of these individual investigations, together with extensive testing, gives the certifying authority the necessary confidence to allow the system to enter general service.

This is, of course, an extreme simplification of the certification process. The design and certification processes often proceed in parallel and there tends to be different levels of certification in the many phases of the development of a complex project like producing a new aircraft.

How can we know the programs are correct?

A computer program embedded in a safety-critical system is just another component in the system. Errors in the program are equivalent to design errors in a hardware component; each replication of the program will produce identical errors. There is no analogue in programming to shelf life or the degradation of performance due to age - programs are ageless, even if the computers that support them can exhibit these random patterns of failure. This is not to say that the programs will remain unchanged - changing operational requirements will require different programs. However, changing the software (or firmware) must be considered to be a design change to the total system and not merely the replacement of a faulty component.

Safety-critical programs actually in operation have obviously been subjected to as much scrutiny as the hardware parts of the systems in which they are embedded. These scrutinies generally have not been highly formal - the tools and languages used to produce the programs do not lend themselves to formal methods. Note that introducing a formalism does not in itself improve the situation; the important thing is, in some sense, the "obviousness" of the proof of the program. For example, I would feel very dubious about a program which could **only** be proved with the assistance of a sophisticated theorem-prover: who proves the theorem-prover? On the other hand, it is clear that appropriate formalism in the specifications, languages, tools and computers used, can make verification much easier by preserving this "obviousness" of proof through its various transformations from specifications to code in roms.

In the last analysis, the thing that has to be verified is that the program code embedded in the store of the on-line computer matches the external specifications defining the job that computer is intended to perform. The process of actually producing this code may go through many different refinements from its specification to the raw binary in memory. Of course, these refinements can only be considered to be way-marks in a reasonable path for determining the correctness of the code. As an example, it is not sufficient to verify that a program text written in some language satisfies the external specifications; one also must satisfy oneself that the code produced by the compiler for this language actually matches the text. If the language was a simple assembly language, one could conceive that the assembler program itself could be verified to give this last step. On the other hand, I don't think that I would believe a "verification" of a compiler for a high level language of any great complexity - the size of the compiler is likely to be just too big. Similarly, proofs of large program support systems are likely to be thin on the ground; I would require convincing that the configuration control of a multi-compilation unit is in fact operating correctly. In other words, I would never trust a compiler or assembly system unconditionally; one

requires to be able to look at both the total text and some assembler level representation to be certain that code matches the high-level text and that the text is what one thinks it is. This places certain constraints on both the language and processor used in the safety-critical system if one is to have confidence in this matching. The remainder of this paper is devoted to looking at these constraints.

What kind of programs are critical?

The use of any language on any machine represents a balancing act between the difficulty of writing the program in the language and the complexity of implementation of the compiler for that language on that machine. Thus, the gain derived by having a "verified" assembler is lost by the opacity of all but the simplest programs written in assembly code. On the other hand, using a very high level language means that one has to have an almost religious belief that the compiler writer has got it right; the transformations and mappings made by the compiler are likely to be very difficult to follow, even for the compiler writer. By the same token, an excessively complicated order code in the processor would raise similar worries about whether the chip designer or the micro-coder has got it right in all cases.

In order to determine an appropriate language for safety-critical systems, it is necessary to determine the kind of programs used in them. It is certainly quite easy to give examples of programs that one does **not** want to program in the language; these include compilers, general purpose operating systems, relational databases, program support environments etc. Furthermore, the notion of transportability is quite foreign. A safety-critical program is usually the only program in the particular computer embedded in the system; as such its only relevance is in the particular environment of that system. The programming tools required to construct and analyse safety-critical programs are quite different in kind from these programs and certainly will run in an unrelated computer with an unrelated operating system. This, of course, could be one of the trendy transportable program support environments.

Timing constraints form an important part in the specifications of most safety-critical programs: sensors must be read and controls must be serviced within given time intervals. A large part of any certification exercise is concerned with ensuring that no possible path through the program contravenes these constraints. If the programming language permits arbitrary branching or recursion this task can be very difficult, even with sophisticated program analysis tools. This suggests that the language used should be primitive recursive so that an upper bound can be put to the time required to

evaluate **any** construction in the language. This might seem to be a terrible restriction to those used to the freedom of writing non-critical programs in standard languages. However, in all the safety-critical applications that I know about, informal rules and programming restrictions are applied to the language used to give exactly the effect of it being a primitive recursive language. Indeed some applications have gone even further in banning all loops other than an outer one.

There would be little point in insisting on this accountablity for time if the processor allowed unrestricted interrupts. Similarly, the transput instructions should not be used unless the processor can transmit the data without an indefinite wait. As well as messing up any calculation of evaluation times, interrupts play the devil with any proof, formal or informal, of the program. Indeed, any parallel processing using common memory, is fraught with dangers and difficulties - it is too easy to make mistakes in the interaction between different processes and too difficult to prove that one has not made them. Hence, our safety critical program consists of a single process, polling its input and output, and any time constraints can be verified by the fact that the process is the evaluation of a primitive recursive function.

The Orwellian solution - bounds on space and time

The restriction of language to limit one's capacity for dangerous thought has already been predicted for this year (1984) in a somewhat different context [1]. In safety-critical programming, the attractive freedoms of recursion or unrestricted branching and looping are definitely subversive thoughts. Thus, computer NewSpeak must be a language in which such thoughts are not only forbidden, but unthinkable; the job of the ThoughtPolice (in the shape of the certifying authority) then becomes much easier, not to say feasible. The restriction of primitive recursion means, for example, that the OldSpeak terms GOTO and LABEL do not exist in NewSpeak, either in its text or semantics. This frees the ThoughtPolice from the sometimes quite difficult task of ensuring that the branching via GOTOs and LABELs do not violate timing constraints; there is also the added bonus that GOTOless programming is generally much easier to understand and prove. The efforts of the ThoughtPolice can therefore be directed at more subtle, and hence more dangerous, forms of subversion.

Just as the time taken by a program should be bounded in NewSpeak, so should the space required for the program and its data. This implies a static storage allocation scheme for the compilation of NewSpeak. This in turn means a simple correspondence can be made between names in NewSpeak and addresses in the compiled code. This gives

considerable help in seeing whether the compiled code is a correct mapping of the NewSpeak text. In NewSpeak itself, this means that dynamic sized arrays and generated heap storage are ruled out; recursive procedures, which also require dynamic storage allocation, are already ruled out by the primitive recursive restriction. Similar considerations rule out procedural values eg procedures used as parameters to other procedures.

The Orwellian solution - bounds on arithmetic

As with space and time, all of the numbers handled in NewSpeak are similarly bounded and are members of finite sets. It must be unthinkable that a NewSpeaker might confuse an element of a finite integral range with a mathematical integer or a floating point number with a mathematical real. This sounds so obvious that it hardly needs saying! However, in every language that I know of, the expression x+1 has the same mode as x - which implies that the mode of x describes an infinite set in any reasonable interpretation of the symbols. Most language definitions get over this by allowing the possibility of dynamically produced exceptional values if the result of the operation is outside some range. Different languages treat these exceptional values in different ways: Pascal [2], together with most other languages, simply says that they are illegal while Ada™ [3] allows them to be trapped in the language. The Ada approach is roughly equivalent to a cross between GOTOs and interrupts, both of which are anathemas to the ThoughtPolice and should be unNewSpeakable. The other approach is no better, since what is at fault is not the treatment of exceptional values, but the fact they can be produced at all.

One can usefully define the generic arithmetic operations like +, -, * etc so that they never produce exceptional values by simply using "worst-case" rules about the relationship between their operands and their results. For example, if one knows that x is an integer in the range [0..9] then x+1 is obviously an integer in the range [1..10] and 2*x is in the range [0..18]. If the mode of an integer value in NewSpeak always describes a finite subset of the integers to which the value must belong, then one can think of "worst-case" rules of greater or less complexity to define the integer operations. A simple rule might be that the mode of the result of an operation is the smallest contiguous range that must contain the result. More complicated strategies might also be considered. For example, it is clear that divide can never be defined with a divisor belonging to a set which includes zero; hence, it might be useful for the compiler to be able to deduce that integer expressions like 2*x+1 can never be zero. This could be done by defining + and * so that their result modes are the smallest sets that contain their results. The criteria for choosing which rules are to apply must be influenced by how easy it is

to implement the mode algorithms in the compiler and how much confidence one could place on their correctness.

In this discussion on arithmetic operations, I have assumed that the language uses the modes of the operands only to deduce the mode of the result and not the particular operands themselves. For example, the expression (x-5)*(x-5) is clearly not negative; however to deduce this it is necessary to notice that the two operands of * are equal - the "worst-case" rules on the modes given above would admit negative values in the result. I think it would be going too far to take advantage of this kind of correlation between the operands for two reasons. First, I don't know where to stop - in principle all the theorems on the arithmetic of finite sets of integers are available. Particular operations used commonly like square could be provided as primitives. Second, this kind of comparison between expressions can be error prone and also has the effect of producing highly "discontinuous" mappings between text and compiled code - small changes in the text can produce dramatic changes in code, as anybody who has used a highly optimising compiler can attest. This discontinuity would be very disconcerting to the ThoughtPolice in the development and investigation of a NewSpeak program.

The Orwellian solution - assertions and conditions

Most programs, in NewSpeak or not, will require for their legal operation that some conditions are asserted which cannot be deduced automatically by the compiler. An example of this could be the assignment of a value in one range to a variable of a smaller range - the actual value would, of course, have to be in the smaller range. Indexing arrays also provide examples of this in most languages. It must be an important principle in NewSpeak that such an assertion is never assumed implicitly - the assertion must be writ large and ideally the author of its proof (done independently) identified. Wherever possible, the ThoughtPolice prefer belt and braces; in other words, the assertions should be tested dynamically. If the assertion fails, there is either a hardware error or its proof is faulty. In the latter case, the author of the proof can expect a call from the ThoughtPolice. In either case, the program is untrustworthy and all of its outputs must be considered suspect. It would be nice if the processor were stopped dead until further external action were taken - this would at least give some opportunity to find what is at fault, although this might be of little consolation if one were halfway across the Atlantic at the time.

Of course, it would be intolerable if every second statement in NewSpeak was such an assertion; the language should be designed that its normal conditional structure can aid the compiler to make useful deductions by inheriting information about the values being

manipulated. In this respect, the most useful kind of conditions, whether in assertions or in normal control flow testing, are those which restrict the mode of an expression eg by showing that a value is in a given fixed range. The compiler (and a prover) is then in a position to make futher deductions in the use of this value. As an example, it is easier to extract useful information from a case construction testing for x to be in the range [0..10] than to try and extract the same meaning from a conditional using a boolean expression x>=0 AND x<=10. Other things being equal, case constructions usually can impart more useful information into the evaluation of its various alternative arms than the equivalent conditionals. This is true even of the simple case constructions of Ada and Pascal where the value under consideration must effectively be an integer. A generalisation of case clauses is one which has a control value of more general mode which is discriminated by having a different restricted mode in each arm of the case clause. This generalisation enormously increases the power of the case construction without detracting from its "obviousness" in either text or code. As an example, a structured value could be classified into several sub-modes depending of the value of its fields, thus giving an obvious way of testing the ranges of two values in parallel. If, in addition, one can name the value with the restricted mode for the extent of a particular arm, the inheritance of this restricted mode is made more obvious and easier to use.

The Orwellian solution - expressions and modes

This last sentence perhaps betrays my preference for expression languages rather than statement languages - I prefer to name a value rather than be obliged to create a variable to contain it. This preference is not entirely a question of taste since if I give a name to a value, I know that the value corresponding to this name will not change over the range of the name - something that I might have to work very hard to prove if I had had to create a variable. In fact, most of the trouble in proving that a program text meets its specification arises from the use of variables and assignments. The other constraints on NewSpeak make it impossible to do without them but their use should be as limited as possible. For example, the aliasing of references or variables, eg as parameters of procedures, seem to raise all sorts of difficulties, far out of proportion to their usefulness. For this reason, aliasing is unthinkable in NewSpeak, eg procedures should take only value parameters, with no weasel words about efficiency of implementation as in Ada. Similarly, OUT parameters are just a confusing way of delivering a multiple value as a result to a procedure; the facility it provides is met more generally, and less confusingly, by providing more flexible ways of assigning and naming sub-values, independently of procedures.

All loops in NewSpeak must be bounded. For example, the control variable of a for-statement goes through the elements of a finite, fixed set and, of course, has a mode defined by exactly this set. Classically, for-statements could only ever do anything by side-effects; if there are no assignments in the body of a loop, then the for- statement is null since one could not deliver a result from it. However, there are many stereotypes of loops for which this is annoying and confusing. For example, many loops just create new multiple values; these would be more understandable as explicit replications of values or image calls of procedures. Similarly, standard arithmetic operations like inner product might very well be operations in the language.

It is taken as granted that NewSpeak is a strongly moded language. Indeed, I would like to express this more strongly since, in a sense, the term has been debased. Most languages described as "strongly moded" have implicit illegal values or exceptions so that, for example, the operation, +, can deliver an exceptional value in place of the normal arithmetic answer on overflow. The "mode" of this exceptional value never appears explicitly and does not enter into the deliberations of the compiler in determining the legality or otherwise of the program. These illegal values and exceptions crop up in different languages in all sorts of constructions and forms. Another common one comes from indexing an array outside its bounds; others include some manipulations of variant records, use of uninitialised variables etc. NewSpeak must eschew such looseness in its mode structure - if a construction compiles then it must give its expected answer as defined by the mode rules without exception. Much of this looseness can be avoided, in any case, in exceedingly simple ways, eg there is no reason in the world why variable declarations should not **always** be initialised. Note that this is **not** the same as saying that there is a default value for unset variables. Defaults in general should be deprecated; they are a source of a great deal of errors and add little or nothing to the expressive power of the language.

So far, I have been rather vague about the types of data which can be NewSpeaked. The integral modes are described by fixed, finite subsets of the mathematical integers. Reals present a special problem since floating-point representations and operations require accuracy, as well as range, constraints to be applied and furthermore one cannot usually test dynamically that an accuracy constraint holds. Abstract sets are possible; however they should **be** abstract in that one cannot insist on a particular representation for them. This relieves much of the pressure on the overloading of denotation names felt by Ada, for example. Cartesian products of values give structured or arrayed (or rather indexable-structured) modes with appropriate selectors or indices. The Cartesian sum of values give unions with appropriate means of injection and selection - doing this properly avoids the nastinesses and exceptions inevitably associated with Pascal and Ada

variant structures. Any value can be named, any value can be assigned to a variable of the apropriate mode and any value can be a parameter to, or a result of, a properly formed procedure. Neither variables nor procedures are themselves values for reasons discussed above, although of course, they do have names.

Conclusion

It will be clear from the preceding paragraphs that I do have a particular language in mind with syntax and semantics defined as a model for NewSpeak. However, a particular syntax is a relatively unimportant part of a language. The syntax is only the hook on which is hung the semantics and the semantics are a formalisation of the underlying philosophy of the language. Hence, I have been careful not to give any examples of my NewSpeak text so that the understanding, and hopefully, the acceptance of the philosophy is not confused by differing tastes in syntax.

I believe that the acceptance of the principles outlined above will significantly aid the construction and certification of safety-critical programs. The part played by these programs will form an increasing part of the safety-critical systems of the future and we are likely to have far more automatic safety-critical systems - more and more lives will depend on them.

I would leave you with this final thought: would you buy a "drive-by-wire" car controlled by a program which did **not** obey these principles?

References

1. Nineteen eighty four : G Orwell, first published in 1949.

2. Specification for Pascal : British Standards Institute BS 6192:1982

3. Ada[TM] Reference Manual : DoD Mil-Std 1815A

DOCUMENT CONTROL SHEET

Overall security classification of sheet .....UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Memorandum 3924	3. Agency Reference	4. Report Security Classification U/C
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location ROYAL SIGNALS AND RADAR ESTABLISHMENT		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title
ORWELLIAN PROGRAMMING IN SAFETY-CRITICAL SYSTEMS

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
CURRIE I F				

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement
UNLIMITED

Descriptors (or keywords)

continue on separate piece of paper

Abstract Computers are being used increasingly in situations where lives will be at hazard if they fail. Failures resulting from software errors are both avoidable and unforgivable. This paper is concerned with the final refinement in the production of a critical program from an elementary language via a compiler to bits in some rom in the critical system. It argues that existing languages are inherently unsuitable for this purpose since they allow freedoms of expression and thought which, although convenient in some applications, lead to misconceptions and obscurities. Orwellian programming should limit one's freedom of expression so that these heretical tendencies are unthinkable by using a NewSpeak which permits only Good Thoughts.

This paper was originally given at a conference sponsored by the International Federation of Information Processing and the International Federation of Operational Research Societies on "System Implementation Languages, Experience and Assessment" held at Canterbury University in August 1984.

END

DTIC

7-86